HLA Compile-Time Functions Appendix H

H.1 Conversion Functions

The conversion functions translate data from one format to another. For example, functions in this group can convert integers to strings or strings to integers. These routines provide the compile-time equivalent of the HLA Standard Library CONV module.

The compile-time conversion routines are unusual in the set of compile-time function insofar as they do not require a leading "@" symbol. Instead, the conversion routines use the names of several of the built-in data types. The following table describes each of these functions:

Table 1: Compile-Time Data Conversion Functions

Function	Parameters[a]	Description
boolean	boolean(*constExpr*) ConstExpr can be a boolean, integer, character, or string operand.	If constExpr is numeric, this function returns false for zero and true for any other value. For characters, "t" or "f" returns true or false (respectively), anything else is an error. For strings, the operands must be "true" or "false" (else an error occurs). The boolean function returns boolean values unchanged and returns an error for any other type.
int8 int16 int32 uns8 uns16 uns32 byte word dword	*xxxx*(*constExpr*) Note: *xxxx* represents one of the function names to the left. *constExpr* can be any constant expression that evaluates to a numeric, character, boolean, or string operand.	These functions will convert their operand to the specified data type. These functions generate an error if the resulting value will not fit in the specified data type (e.g., int8(-1000) will generate an error). Note that HLA treats *byte*, *word*, and dword functions identically to *uns8*, *uns16*, and *uns32* (respectively). For boolean operands, true returns one and false returns zero. If the operand is a real value, then these functions truncate the value to obtain the corresponding integer return value. For character operands, these function return the corresponding ASCII code of the character. For string operands, the string must be a legal sequence of characters that form a decimal number.

Table 1: Compile-Time Data Conversion Functions

Function	Parameters[a]	Description
real32 real64 real80	*xxxxxx*(*constExpr*) Note: *xxxxxx* represents one of the function names to the left. *constExpr* can be any constant expression that evaluates to a numeric or string operand.	These functions convert their specified operand to the corresponding real value. These functions convert integer operands to the corresponding real value. If the operand is a string expression, it must be a valid sequence of characters that corresponds to an HLA floating point value. These functions convert that string to the corresponding real value.
char	char(*constExpr*) *constExpr* must be a positive integer value, a character, or a string.	If the parameter is an integer value, this function returns the character with that ASCII code. If the parameter is a string, this function returns the first character of that string. If the operand is a character, this function simply returns that character value.
string	string(*constExpr*) *constExpr* can be any legal constant data type.	This function returns the string representation of the specified parameter. For real values, this function returns the scientific notation format for the value. For boolean expressions, this function returns the value "true" or "false". For character operands, this function returns a string containing the single character specified as the operand. For integer parameters, this function returns a string containing the decimal equivalent of that value. For character set operands, this function returns a string listing all the characters in the character set. If the operand is a string expression, this function simply returns that string.
cset	cset(*constExpr*) *constExpr* can be a character, string, or a cset.	This function returns a character set containing the characters specified by the operand. If the operand is a character, then this function returns the singleton set containing that single character. If the operand is a string, this function returns the union of all the characters in that string. If the operand is a character set, this function returns that character set.

Table 1: Compile-Time Data Conversion Functions

Function	Parameters[a]	Description
text	text(*constExpr*) *constExpr*: same as for string.	This function takes the same parameters as the string function. Instead of returning a string constant, however, this function expands the text in-line at the point of the function. This function is equivalent to the *@text* function.
@odd	@odd(*constExpr*) *constExpr* must be an integer value.	This function returns true if the integer operand is an odd number, it returns false otherwise.

a. Integer operands can be any of the *intXX, unsXX, byte, word*, or *dword* types.

Internally, HLA generally maintains all numeric constant values as *int32, uns32, dword*, or *real80*. Therefore, it is unlikely that you would want to use the *int8, int16, uns8, uns16, real32*, or *real64* compile-time functions in your program unless you want to force an error if a value is out of range.

Of these conversion functions, the *string* function is, perhaps, the most useful. Many compile-time functions and statements accept only string operands. You can use the *string* function to translate other data types to a string in a situation where you wish to use one of these other data types. For example, the #ERROR statement only allows a single string parameter. However, you can construct complex error messages by using the string concatentation operator ("+") with the *string* function, e.g.,

```
#error( "Constant value i32=" + string(i32) + " and that is out of range" )
```

H.2 Numeric Functions

These functions provide common mathematical functions. Remember, these functions compute their values at compile-time. Their parameters are constants and they return constant values. These functions are not useable with variables at run-time. See the HLA Standard Library for comparable functions you can call from your assembly language programs while they are running.

Table 2: Numeric Compile-Time Functions

Function	Parameters[a]	Description
@abs	@abs(*constExpr*) *constExpr* must be a numeric value.	Returns the absolute value of the parameter. The return type is the same as the parameter type.

Table 2: Numeric Compile-Time Functions

Function	Parameters[a]	Description
@byte	@byte(*Expr*, *select*) *Expr* must be a 32-bit ordinal expression, a real expression (32, 64, or 80 bits), or a *cset* expression. *select* must be less than the size of *Expr's* data type.	This function exacts the specified byte from the value of *Expr* as selected via the *select* parameter. If *select* is zero, this function returns the L.O. byte, higher values for select return the corrsponding higher-order bytes of the object. Note that HLA usually extends literal constants to the largest representation possible, e.g., HLA treats 1.234 as a real80 value. Use coercion to change this, if necessary (e.g., @byte(real32(1.234), 3))
@ceil	@ceil(*constExpr*) *constExpr* must be a numeric value.	If the parameter is an integer value, this function simply converts it to a real value and returns that value. If the parameter is a real value, then this function returns the smallest integer value larger than or equal to the parameter's value (i.e., this function rounds a real value to the next highest integer if the real value contains a fractional part).
@cos	@cos(*constExpr*) *constExpr* must be a numeric value expressing an angle in radians.	This function returns the cosine of the specified parameter value.
@exp	@exp(constExpr) *constExpr* must be a numeric value.	This function return *e* raised to the specified power.
@floor	@floor(*constExpr*) *constExpr* must be a numeric value.	This function returns the largest integer value that is less than or equal to the numeric value passed as a parameter. For positive real numbers this is equivalent to truncation (for negative numbers, it rounds towards negative infinity).
@log	@log(*constExpr*) *constExpr* is a non-negative numeric value.	This function computes the natural (base *e*) logarithm of its operand.

Table 2: Numeric Compile-Time Functions

Function	Parameters[a]	Description
@log10	@log10(*constExpr*) *constExpr* is a non-negative numeric value.	This function computes the base-10 logarithm of its operand.
@max	@max(*list*) *list* is a list of two or more comma separated numeric expressions.	This function returns the maximum of a set of numeric values. The values in the list must all be the same type.
@min	@min(*list*) *list* is a list of two or more comma separated numeric expressions.	This function returns the minimum of a set of numeric values. The values in the list must all be the same type.
@random	@random(*intExpr*) *intExpr* must be a positive integer value.	This function returns a pseudo-random number between zero and *intExpr-1*. Currently HLA uses the random function provided by the C standard library; so don't expect a high quality random number generator here. In particular, if *intExpr* is a small value, the quality of the random number generator is very low.
@randomize	@randomize(*intExpr*) *intExpr* must be a positive integer value.	This function attempts to "randomize" the random number generator seed. Then it returns a random number between zero and *intExpr-1*. You should not call this function multiple times in your source file.
@sin	@sin(*constExpr*) *constExpr* must be a numeric value expressing an angle in radians.	This function returns the sine of the specified parameter value.
@sqrt	@sqrt(constExpr) *constExpr* m ust be a non-negative numeric value.	This function returns the square root of the specified operand value.

Table 2: Numeric Compile-Time Functions

Function	Parameters[a]	Description
@tan	@tan(*constExpr*) *constExpr* must be a numeric value expressing an angle in radians.	This function returns the tangent of the specified parameter value.

a. Numeric parameters are *intX, unsX, byte, word, dword,* or *realX* values.

H.3 Date/Time Functions

The Date/Time compile-time functions return strings holding the current date and time.

Table 3: HLA Compile-Time Date/Time Functions

Function	Parameters	Description
@date	@date	This function returns a string specifying the current date. This string typically takes the form "year/month/day", e.g., "2000/12/31".
@time	@time	This function returns a string specifying the current time. This string typically takes the form "HH:MM:SS xM" (x= A or P).

H.4 Classification Functions

These functions test a character or string to see if the characters belong to a certain class (e.g., alphabetic characters). This functions return true or false depending upon the result of the comparison.

If the operand is a character expression, these functions return true if that character belongs to the specified class. If the operand is a string, these functions return true if all characters in the string belong to the specified class.

Also see the HLA Compile-Time Pattern Matching Functions for some different ways to test characters in a string. Remember, these are compile-time functions. The HLA Standard Library contains comparable routines for use at run-time in your programs. See the HLA Standard Library documentation for more details.

Table 4: HLA Compile-Time Character Classification Functions.

Function	Parameters	Description
@isalpha	@isalpha(*constExpr*) *constExpr* must be a character or a string expression.	This function returns true if the character operand is an alphabetic character. If the parameter is a string, this function returns true if all characters in the string are alphabetic.
@isalphanum	@isalphanum(*constExpr*) *constExpr* must be a character or a string expression.	This function returns true if the character operand is an alphanumeric character. If the parameter is a string, this function returns true if all characters in the string are alphanumeric.
@isdigit	@isdigit(*constExpr*) *constExpr* must be a character or a string expression.	This function returns true if the character operand is a decimal digit character. If the parameter is a string, this function returns true if all characters in the string are digits.
@islower	@islower(*constExpr*) *constExpr* must be a character or a string expression.	This function returns true if the character operand is a lower case alphabetic character. If the parameter is a string, this function returns true if all characters in the string are lower case alphabetic characters.
@isspace	@isspace(*constExpr*) *constExpr* must be a character or a string expression.	This function returns true if the character operand is a space character[a]. If the parameter is a string, this function returns true if all characters in the string are spaces.
@isupper	@isupper(*constExpr*) *constExpr* must be a character or a string expression.	This function returns true if the character operand is an upper case alphabetic character. If the parameter is a string, this function returns true if all characters in the string are upper case alphabetic characters

Table 4: HLA Compile-Time Character Classification Functions.

Function	Parameters	Description
@isxdigit	@isxdigit(*constExpr*) *constExpr* must be a character or a string expression.	This function returns true if the character operand is a hexadecimal digit character {0-9, a-f, A-F}. If the parameter is a string, this function returns true if all characters in the string are hexadecimal digits.

a. "Space" means any white space character. This includes tabs, newlines, etc.

H.5 String and Character Set Functions

The following functions provide a very powerful set of string manipulation functions to the HLA compile-time language. These functions let you easily maniipulate data like macro parameters and #text..#endtext blocks.

Remember, these are compile-time functions. The HLA Standard Library contains comparable routines for use at run-time in your programs. See the HLA Standard Library documentation for more details.

The @extract function behaves a little differently than the *cs.extract* function in the HLA Standard Library. @extract does not actually remove the specified character from the character set. Keep this in mind if you use both @extract and *cs.extract* frequently.

Table 5: HLA Compile-Time String Functions

Function	Parameters	Description
@delete	@delete(*strExpr*, *start*, *len*) strExpr must be a string expression. *start* and *len* must be positive integer expressions.	This function returns a string consisting of the strExpr parameter with *len* characters removed starting at position *start* in the string. The first character in the string is at position zero. Therefore, @delete("Hello", 2, 3) returns the string "He".

Table 5: HLA Compile-Time String Functions

Function	Parameters	Description
@extract	@extract(*csetExpr*) *csetExpr* must be a character set value.	This function returns an arbitrary character from the specified character set. Note that this function does not remove that character from the set. You must manually remove the character if you do not want the next call to @extract (with the same parameter) to return the same character, e.g., val c := @extract(someSet); someSet := someSet - {c};
@index	@index(*strExpr, start, findStr*) *strExpr* and *findStr* must be string expressions. *start* must be a non-negative integer value.	This function searches for the string specified by *findStr* within the *strExpr* string starting at character position *start*. If this function finds the string, it returns the index into *strExpr* where it located *findStr*. If it does not find the string, it returns -1.
@insert	@insert(*destStr, start, strToIns*) *destStr* and *strToIns* must be string expressions. *start* must be a non-negative integer expression.	This function returns a string consisting of the combination of the *destStr* and *strToIns* strings. This function inserts *strToIns* into *destStr* at the position specified by *start*.
@length	@length(*strExpr*) *strExpr* must be a string expression.	This function returns the length of the specified string as an integer result.
@lowercase	@lowercase(*strExpr, start*) *strExpr* must be a string expression. *start* must be a non-negative integer expression.	This function translates all characters from position *start* to the end of the string to lower case (if they were previously upper case alphabetic characters).

Table 5: HLA Compile-Time String Functions

Function	Parameters	Description
@rindex	@rindex(*strExpr*, *start*, *findStr*) *strExpr* and *findStr* must be string expressions. *start* must be a non-negative integer value.	This function searches backwards to find the last occurrence of the string specified by *findStr* within the *strExpr* string. This function will only search back to position *start*. If this function finds the string, it returns the index into *strExpr* where it located *findStr*. If it does not find the string, it returns -1.
@strbrk	@strbrk(*strExpr, start, csetExpr*) *strExpr* must be a string expression. *start* must be a non-negative integer expression. *csetExpr* must be a character set expression.	Starting at position *start* within *strExpr*, this function searches for the first character of *strExpr* that is a member of the *csetExpr* character set. It returns -1 if no such characters exist.
@strset	@strset(*charExpr, len*) *charExpr* must be a character value. *len* must be a non-negative integer value.	This function returns the string consisting of *len* copies of *charExpr* concatenated together.
@strspan	@strspan(*strExpr, start, csetExpr*) *strExpr* must be a string expression. *start* must be a non-negative integer expression. *csetExpr* must be a character set expression.	Starting at position *start* within *strExpr*, this function searches for the first character of *strExpr* that is not a member of the *csetExpr* character set. It returns the length of the string if all remaining characters are in the specified character set.
@substr	@substr(*strExpr, start, len*) *strExpr* must be a string expression. *start* and *len* must be non-negative integer values.	This function returns the sequence of *len* characters (the "substring") starting at position *start* in the *strExpr* string.

Table 5: HLA Compile-Time String Functions

Function	Parameters	Description
@tokenize	@tokenize(*strExpr, start, Delims, Quotes*) *strExpr* must be a string expression. *start* must be a non-negative integer value. *Delims* and *Quotes* must be character set expressions.	This function returns an array of strings consisting of the various substrings in *strExpr* obtained by separating the substrings using the *Delims* character set. This "lexical scan" operation begins at position *start* in *strExpr.* The function first skips over all characters in *Delims* and the collects all characters until it finds another delimiter character from the *Delims* set. It repeats this process, adding each scanned string to the array it returns, until it reaches the end of the string. The fourth parameter, *Quotes*, specifies special quoting characters. Typically, this character set is the empty set. However, if it contains a character, then that character is a quote character and all characters between two occurrences of the quote symbol constitute a single string, even if delimiters appear within that string. Typical characters in the *Quotes* character set would be apostrophes or quotation marks. If "(", "[", or "{" appears in the *Quotes* set, then the corresponding closing symbol must also appear and tokenize uses the pair of quote objects to surround a quoted item. This function is unusual insofar as it returns an array constant as its result. Typically you would assign this to a VAL or CONST object so you can gain access to the individual items in the array. To determine the number of elements in this array, use the @elements function.

Table 5: HLA Compile-Time String Functions

Function	Parameters	Description
@trim	@trim(*strExpr*) *strExpr* must be a string expression.	This function returns the specified string operand with leading and trailing spaces removed.
@uppercase	@uppercase(*strExpr*) *strExpr* must be a string expression.	The function returns a copy of its string operand with any lower case alphabetic characters converted to upper case.

H.6 Pattern Matching Functions

The HLA compile-time language provides a large set of pattern matching functions similar to the run-time functions available in the PATTERNS.HHF module of the HLA Standard Library. These pattern matching functions, combined with the string processing functions of the previous section, provide you with the ability to write your own scanners and parsers (i.e., compilers) within the HLA compile-time language. Indeed, one of the primary purposes of the HLA compile-time language is to let you expand the HLA language to suit your needs. The pattern matching functions, along with macros, provide the backbone for this capability in HLA.

Note that there are some pretty serious differences between the HLA compile-time pattern matching functions and the routines in the PATTERNS.HHF module. Perhaps the biggest difference is the fact that the compile-time functions do not support backtracking while the run-time routines do. Fortunately, it is not that difficult to simulate backtracking on your own within the compile-time language.

Another difference between the comile-time functions and their run-time counterparts is the parameter list. The compile-time functions typically have a couple of optional parameters that let you extract information about the pattern match if it was successful. Consider, for a moment, the @matchStr function:

```
@matchStr( Str, tstStr )
```

When you call this function using the syntax above, this function will return true if the string expression *Str* begins with the sequence of characters found in *tstStr*. It returns false otherwise. Now consider the following invocation:

```
@matchStr( Str, tstStr, remainder )
```

This call to the function also returns true if *Str* begins with the characters found in the *tstStr* string. If this function returns true, then this function also copies the remaining charactes (i.e., those characters in *Str* that following the *tstStr* characters at the beginning of *Str*) to the *remainder* VAL object. If @matchStr returns false, the value of *remainder* is undefined (with the single exception noted below).

It is perfectly legal to specify the same string as the *Str* and *remainder* parameters. For example, consider the following invocation:

```
@matchStr( str, "Hello ", str )
```

If *str* begins with the string "Hello " then this function will return true and replace *str's* value with the string containing all characters beyond the sixth character of the string. If *str* does not begin with "Hello " then this function does not modify *str's* value.

Most of the HLA compile-time pattern matching functions also allow a second optional parameter. Consider the following invocation of the @oneOrMoreCset function:


```
@oneOrMoreCset( str, {'0'..'9'}, remainder, matched )
```

If this function returns true it will copied the sequence of characters at the beginning of *str* that are members of the specified character set (i.e., digits) to the *matched* VAL object. This function returns all characters beyond the digits in the *remainder* VAL object. If this function returns false, then *remainder* and *matched* will contain undefined values and this function will not affect *str*.

H.6.1 String/Cset Pattern Matching Functions

Among the most useful of the pattern matching functions are those that checking the leading characters of a string to see if they are members of a particular character set. The following rich set of functions provides this capability in the compile-time language.

Table 6: HLA Compile-time Cset Pattern Matching Functions

Function	Parameters	Description
@peekCset	@peekCset(*str, cset, rem, matched*) *str* must be a string expression. *cset* must be a character set expression. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The types of *rem* and *matched* are irrelevant but they must be VAL objects. This function stores a result into these two objects.	This function returns true if the first character of *str* is a member of *cset*; it returns false otherwise. If *rem* is present, this function copies *str* to *rem* upon return. If *matched* is present and the function returns true, this function stores a copy of the first character into the *matched* VAL object. The value of *matched* is undefined if this function returns false.

Table 6: HLA Compile-time Cset Pattern Matching Functions

Function	Parameters	Description
@oneCset	@oneCset(*str, cset, rem, matched*) *str* must be a string expression. *cset* must be a character set expression. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The types of *rem* and *matched* are irrelevant but they must be VAL objects. This function stores a result into these two objects.	This function returns true if the first character of *str* is a member of *cset*; it returns false otherwise. If *rem* is present, this function copies all characters of *str* beyond the first character to *rem* upon return. If *matched* is present and the function returns true, this function stores a copy of the first character into the *matched* VAL object. The value of *matched* is undefined if this function returns false.
@uptoCset	@uptoCset(*str, cset, rem, matched*) *str* must be a string expression. *cset* must be a character set expression. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The types of *rem* and *matched* are irrelevant but they must be VAL objects. This function stores a result into these two objects.	This function copies all characters up to, but not including, a single character from the *cset* parameter. If the *str* parameter does not contain a character in the cset set, this function returns false. If it succeeds, and the matched parameter is present, it copies all characters it matches to the *matched* parameter and it copies all remaining characters to the *rem* parameter (if present).
@zeroOrOneCset	@zeroOrOneCset(*str, cset, rem, matched*) *str* must be a string expression. *cset* must be a character set expression. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The types of *rem* and *matched* are irrelevant but they must be VAL objects. This function stores a result into these two objects.	This function always returns true. This function copies the single character it matches (if any) to the *matched* string and any remaining characters in the string to the *rem* object (assuming *rem* and *matched* appear in the parameter list).

Table 6: HLA Compile-time Cset Pattern Matching Functions

Function	Parameters	Description
@exactlyNCset	@exactlyNCset(*str, cset, n, rem, matched*) *str* must be a string expression. *cset* must be a character set expression. *n* must be a non-negative integer value. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The types of *rem* and *matched* are irrelevant but they must be VAL objects. This function stores a result into these two objects.	This function returns true if the first *n* characters of *str* are members of *cset*. The character at position *n+1* must not be a member of *cset*. If this function returns true, it copies the first *n* characters of *str* to *matched* and copies any remaining characters to *rem* (assuming *rem* and *matched* are present).
@firstNCset	@firstNCset(*str, cset, n, rem, matched*) *str* must be a string expression. *cset* must be a character set expression. *n* must be a non-negative integer value. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The types of *rem* and *matched* are irrelevant but they must be VAL objects. This function stores a result into these two objects.	This function returns true if the first *n* characters of *str* are members of *cset*. The character at position *n+1* may or may not be a member of *cset*. If this function returns true, it copies the first *n* characters of *str* to *matched* and copies any remaining characters to *rem* (assuming *rem* and *matched* are present).

Table 6: HLA Compile-time Cset Pattern Matching Functions

Function	Parameters	Description
@nOrLessCset	@nOrLessCset(*str, cset, n, rem, matched*) *str* must be a string expression. *cset* must be a character set expression. *n* must be a non-negative integer value. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The types of *rem* and *matched* are irrelevant but they must be VAL objects. This function stores a result into these two objects.	This function always returns true. This function matches up to the first *n* characters of *str* are members of *cset*. The character at position *n+1* may or may not be a member of *cset*. If this function returns true, it copies the matching (up to *n*) characters of *str* to *matched* and copies any remaining characters to *rem* (assuming *rem* and *matched* are present).
@nOrMoreCset	@nOrMoreCset(*str, cset, n, rem, matched*) *str* must be a string expression. *cset* must be a character set expression. *n* must be a non-negative integer value. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The types of *rem* and *matched* are irrelevant but they must be VAL objects. This function stores a result into these two objects.	This function returns true if it matches at least *n* characters from *str* in cset. If *rem* and *matched* appear in the parameter list, this function will copy all characters it matches to *matched* and copy any remaining characters into *rem*.

Table 6: HLA Compile-time Cset Pattern Matching Functions

Function	Parameters	Description
@nToMCset	@nToMCset(*str, cset, n, m, rem, matched*) *str* must be a string expression. *cset* must be a character set expression. *n* and *m* must be non-negative integer values. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The types of *rem* and *matched* are irrelevant but they must be VAL objects. This function stores a result into these two objects.	This function returns true if there are at least *n* characters in *cset* at the beginning of *str*. If *rem* and *matched* are present, this function will copy all the characters it matches (up to the m^{th} position) into the *matched* string and copy any remaining characters into the *rem* string. The character at position *m+1* may or may not be a member of *cset*.
@exactlyNToMCset	@ExactlyNToMCset(*str, cset, n, m, rem, matched*) *str* must be a string expression. *cset* must be a character set expression. *n* and *m* must be non-negative integer values. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The types of *rem* and *matched* are irrelevant but they must be VAL objects. This function stores a result into these two objects.	This function returns true if there are at least *n* characters in *cset* at the beginning of *str*. If *rem* and *matched* are present, this function will copy all the characters it matches (up to the m^{th} position) into the *matched* string and copy any remaining characters into the *rem* string. If this function matches *m* characters, the character at position *m+1* must not be a member of *cset* or else this function will return false.
@zeroOrMoreCset	@zeroOrMoreCset(*str, cset, rem, matched*) *str* must be a string expression. *cset* must be a character set expression. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The types of *rem* and *matched* are irrelevant but they must be VAL objects. This function stores a result into these two objects.	This function always returns true. If *rem* and *matched* are present, this function will copy all characters from the beginning of *str* that are members of *cset* to the *matched* string. It will copy all remaining characters to the *rem* string.

Table 6: HLA Compile-time Cset Pattern Matching Functions

Function	Parameters	Description
@oneOrMoreCset	@OneOrMoreCset(*str, cset, rem, matched*) *str* must be a string expression. *cset* must be a character set expression. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The types of *rem* and *matched* are irrelevant but they must be VAL objects. This function stores a result into these two objects.	This function returns true if the first character of *set* is a member of *cset*. If *rem* and *matched* are present, this function will copy all characters from the beginning of *str* that are members of *cset* to the *matched* string. It will copy all remaining characters to the *rem* string.

H.6.2 String/Character Pattern Matching Functions

Though not always as useful as the character set pattern matching functions, the HLA compile-time character matching functions are more efficent than the character set routines when matching single characters.

Table 7: HLA Compile-time Character Matching Functions

Function	Parameters	Description
@peekChar	@peekChar(*str, char, rem, matched*) *str* must be a string expression. *char* must be a character expression. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The types of *rem* and *matched* are irrelevant but they must be VAL objects. This function stores a result into these two objects.	This function returns true if the first character of *str* is equal to *char*. If *rem* and *matched* are present and this function returns true, it also returns *str* in *rem* and *char* in *matched*.

Table 7: HLA Compile-time Character Matching Functions

Function	Parameters	Description
@oneChar	@oneChar(*str, char, rem, matched*) *str* must be a string expression. *char* must be a character expression. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The types of *rem* and *matched* are irrelevant but they must be VAL objects. This function stores a result into these two objects.	This function returns true if the first character of *str* is equal to *char*. If *rem* and *matched* are present and this function returns true, it also returns all the characters *str* beyond the first character in *rem* and *char* in *matched*.
@uptoChar	@uptoChar(*str, char, rem, matched*) *str* must be a string expression. *char* must be a character expression. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The types of *rem* and *matched* are irrelevant but they must be VAL objects. This function stores a result into these two objects.	This function returns true if the *char* exists somewhere in *str*. If *rem* and *matched* are present and this function returns true, it also returns, in *rem*, all the characters in *str* starting with the first instance of *char*. It also returns all the characters up to (but not including) the first instance of *char* in *matched*.
@zeroOrOneChar	@zeroOrOneChar(*str, char, rem, matched*) *str* must be a string expression. *char* must be a character expression. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The types of *rem* and *matched* are irrelevant but they must be VAL objects. This function stores a result into these two objects.	This function always returns true. If *rem* and *matched* are present, it sets *matched* to the matched string (i.e., an empty string or the single character *char*) and it sets *rem* to the characters after the matched character value (the whole string if the first character of *str* is not equal to *char*).

Table 7: HLA Compile-time Character Matching Functions

Function	Parameters	Description
@zeroOrMore-Char	@zeroOrMoreChar(*str, char, rem, matched*) *str* must be a string expression. *char* must be a character expression. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The types of *rem* and *matched* are irrelevant but they must be VAL objects. This function stores a result into these two objects.	This function always returns true. If the first character of *str* does not match *char*, then this function returns the empty string in *matched* and it returns *str* in *rem*. If *str* begins with a sequence of characters all equal to *char*, then this function returns that sequence in *matched* and it returns the remaining characters in *rem*.
@oneOrMoreChar	@oneOrMoreChar(*str, char, rem, matched*) *str* must be a string expression. *char* must be a character expression. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The types of *rem* and *matched* are irrelevant but they must be VAL objects. This function stores a result into these two objects.	This function returns true if the first character in *str* is equal to *char*; otherwise it returns false. If this function returns true, it copies all leading occurrences of *char* into matched and any remaining characters from *str* into *rem*.
@exactlyNChar	@exactlyNChar(*str, char, n, rem, matched*) *str* must be a string expression. *cset* must be a character set expression. *n* must be a non-negative integer value. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The types of *rem* and *matched* are irrelevant but they must be VAL objects. This function stores a result into these two objects.	This function returns true if the first *n* characters of *str* all match *char*. The $n+1^{th}$ character must not be equal to char. If this function returns true, it returns a string of *n* copies of *char* in *matched* and all remaining characters (position *n* and beyond) in *rem*.

Table 7: HLA Compile-time Character Matching Functions

Function	Parameters	Description
@firstNChar	@firstNChar(*str, char, n, rem, matched*) *str* must be a string expression. *cset* must be a character set expression. *n* must be a non-negative integer value. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The types of *rem* and *matched* are irrelevant but they must be VAL objects. This function stores a result into these two objects.	This function returns true if the first *n* characters of *str* all match *char*. The $n+1^{th}$ character may or may not be equal to char. If this function returns true, it returns a string of *n* copies of *char* in *matched* and all remaining characters (position *n* and beyond) in *rem*.
@nOrLessChar	@nOrLessChar(*str, char, n, rem, matched*) *str* must be a string expression. *cset* must be a character set expression. *n* must be a non-negative integer value. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The types of *rem* and *matched* are irrelevant but they must be VAL objects. This function stores a result into these two objects.	This function returns true as long as *str* begins with *n* or fewer (including zero) copies of *char*. It fails if *str* begins with more than *n* copies of *char*. If this function returns true, it returns a string, in *matched*, containing all copies of *char* that appear at the beginning of *str*. It returns all remaining characters in *rem*.
@nOrMoreChar	@nOrMoreChar(*str, char, n, rem, matched*) *str* must be a string expression. *cset* must be a character set expression. *n* must be a non-negative integer value. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The types of *rem* and *matched* are irrelevant but they must be VAL objects. This function stores a result into these two objects.	This function returns true if *str* begins with at least *n* copies of *char*. It returns false otherwise. If this function returns true, then it returns the leading characters that match *char* in *matched* and all following characters in *rem*.

Table 7: HLA Compile-time Character Matching Functions

Function	Parameters	Description
@nToMChar	@nToMChar(*str, char, n, m, rem, matched*) *str* must be a string expression. *cset* must be a character set expression. *n* and *m* must be non-negative integer values. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The types of *rem* and *matched* are irrelevant but they must be VAL objects. This function stores a result into these two objects.	This function matches any string that has at least *n* copies of *char* at the beginning of str. It will match up to *m* copies of *char*. The character at position *m+1* may be equal to *char*, but this function will not match that character. If this function returns true, it returns the string of matched characters in *matched* and any remaining characters in *rem*.
@exactlyNToMChar	@exactlyNToMChar(*str, char, n, m, rem, matched*) *str* must be a string expression. *cset* must be a character set expression. *n* and *m* must be non-negative integer values. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The types of *rem* and *matched* are irrelevant but they must be VAL objects. This function stores a result into these two objects.	This function matches any string that has at least *n* copies of *char* at the beginning of str. It will match up to *m* copies of *char*. The character at position *m+1* must not be equal to *char*. If this function returns true, it returns the string of matched characters in *matched* and any remaining characters in *rem*.

H.6.3 String/Case Insenstive Character Pattern Matching Functions

HLA provides two sets of character matching routines: the previous section described the standard character matching routines, this section presents the case insensitive versions of those same routines.

Table 8: HLA Compile-time Case Insensitive Character Matching Functions

Function	Parameters	Description
@peekiChar	@peekiChar(*str, char, rem, matched*) See @peekChar for details.	Case insensitive version of @peekChar. See @peekChar for details.
@oneiChar	@oneiChar(*str, char, rem, matched*) See @oneChar for details.	Case insensitive version of @oneChar. See @oneChar for details.
@uptoiChar	@uptoiChar(*str, char, rem, matched*) See @uptoChar for details.	Case insensitive version of @uptoChar. See @uptoChar for details.
@zeroOrOneiChar	@zeroOrOneiChar(*str, char, rem, matched*) See @zeroOrOneChar for details.	Case insensitive version of @zeroOrOneChar. See @zeroOrOneChar for details.
@zeroOrMorei-Char	@zeroOrMoreiChar(*str, char, rem, matched*) See @zeroOrMoreChar for details.	Case insensitive version of @zeroOrMoreChar. See @zeroOrMoreChar for details.
@oneOrMoreiChar	@OneOrMoreiChar(*str, char, rem, matched*) See @OneOrMoreChar for details.	Case insensitive version of @OneOrMoreChar. See @OneOrMoreChar for details.
@exactlyNiChar	@exactlyNiChar(*str, char, n, rem, matched*) See @exactlyNChar for details.	Case insensitive version of @exactlyNChar. See @exactlyNChar for details.
@firstNiChar	@firstNiChar(*str, char, n, rem, matched*) See @firstNChar for details.	Case insensitive version of @firstNChar. See @firstNChar for details.
@nOrLessiChar	@nOrLessiChar(*str, char, n, rem, matched*) See @nOrLessChar for details.	Case insensitive version of @nOrLessChar. See @nOrLessChar for details.
@nOrMoreiChar	@nOrMoreiChar(*str, char, n, rem, matched*) See @nOrMoreChar for details.	Case insensitive version of @nOrMoreChar. See @nOrMoreChar for details.

Table 8: HLA Compile-time Case Insensitive Character Matching Functions

Function	Parameters	Description
@nToMiChar	@nToMiChar(*str, char, n, m, rem, matched*) See @nToMChar for details.	Case insensitive version of @nToMChar. See @nToMChar for details.
@exactlyNToMi-Char	@exactlyNToMiChar(*str, char, n, m, rem, matched*) See @exactlyNToMChar for details.	Case insensitive version of @exactlyNToM-Char. See @exactlyN-ToMChar for details.

H.6.4 String/String Pattern Matching Functions

Another set of popular pattern matching routines in the compile-time function set are the string matching routines. These routines check to see if a string begins with some specified sequence of characters. Next to the character set matching functions, these are probably the most commonly used pattern matching functions.

Table 9: Compile-Time String Matching Functions

Function	Parameters	Description
@matchStr	@matchStr(*str, tstStr, rem, matched*) *str* must be a string expression. *tstStr* must be a string expression. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The types of *rem* and *matched* are irrelevant but they must be VAL objects. This function stores a result into these two objects.	This function returns true if *str* begins with the characters found in *tstStr.* If this function returns true, it also returns *tstStr* in *matched* and all characters in str following the *tstStr* characters in *rem*.
@matchiStr	@matchiStr(*str, tstStr, rem, matched*) *Same parameters as matchStr, see that function for details.*	This is a case insensitive version of @matchStr.

Table 9: Compile-Time String Matching Functions

Function	Parameters	Description
@uptoStr	@uptoStr(*str, tstStr, rem, matched*) *Same parameters as matchStr, see that function for details.*	This function returns true if *tstStr* appears somewhere within *str*, it returns false otherwise. If this function returns true, then it returns all characters up to, but not including, the characters from *tstStr* in *matched*. It returns all following characters (including the *tstStr* substring) in *rem*.
@uptoiStr	@uptoiStr(*str, tstStr, rem, matched*) *Same parameters as matchStr, see that function for details.*	This is a case insensitive version of @uptoStr.
@matchToStr	@matchToStr(*str, tstStr, rem, matched*) *Same parameters as matchStr, see that function for details.*	This function is similar to @uptoStr except if it returns true it will copy all characters up to and including *tstStr* to *matched* and all following characters to *rem*.
@matchToiStr	@matchToiStr(*str, tstStr, rem, matched*) *Same parameters as matchStr, see that function for details.*	This is a case insensitive version of @matchToStr.

H.6.5 String/Misc Pattern Matching Functions

This last group of compile-time pattern matching functions check for certain special types of strings, such as HLA identifiers, numeric constants, whitespace, and the end of the string.

Table 10: Miscellaneous Compile-time Pattern Matching Functions

Function	Parameters	Description
@matchID	@matchID(str, rem, matched) *str* must be a string expression. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The types of *rem* and *matched* are irrelevant but they must be VAL objects. This function stores a result into these two objects.	This function returns true if the first sequence of characters in *str* match the definition of an HLA identifier. An HLA identifier is any sequence of characters that begins with an underscore or an alphabetic character that is followed by zero or more underscore or alphanumeric characters. If this function returns true, it copies the identifier to *matched* and all following characters to *rem*.
@matchIntConst	@matchIntConst(str, rem, matched) *str* must be a string expression. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The type of *rem* is irrelevant but it must be a VAL object. If *matched* is present, it must be an *int32* or *uns32* object. **Note: unlike most pattern matching functions, *matched* is not a string object.**	This function returns true if the leading characters of *str* are decimal digits (as per HLA, underscores are legal in the interior of the integer string). If this function returns true, it converts the matched integer string to integer form and stores this value in *matched*. This function returns the remaining characters after the numeric digits in *rem*.

Table 10: Miscellaneous Compile-time Pattern Matching Functions

Function	Parameters	Description
@matchRealConst	@matchRealConst(str, rem, matched) *str* must be a string expression. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The type of *rem* is irrelevant but it must be a VAL object. If *matched* is present, it must be an *real80* object. **Note: unlike most pattern matching functions, *matched* is not a string object.**	This function returns true if the leading characters of *str* correspond to the HLA definition of a real literal constant. If this function returns true, it also returns the remaining characters in *rem* and it converts the real string to *real80* format and stores this value in *matched*.
@matchNumericConst	@matchNumericConst(str, rem, matched) Same parameters as @matchIntConst or @matchRealConst; see those functions for details.	This is a combination of @matchIntConst and @matchRealConst. If this function matches a string at the beginning of *str* that is a legal numeric constant, it will convert that value to numeric form (*int32* or *real80*) and store the value into *matched*. This function returns any following characters in *rem*.
@matchStrConst	@matchStrConst(str, rem, matched) Same parameters as @matchID; see @matchID for details.	This function returns true if *str* begins with an HLA compatible literal string constant. If this function matches such a constant, it will store the matched string, minus the delimiting quotes, into the *matched* variable. It will also store any following charcters into *rem*.

Table 10: Miscellaneous Compile-time Pattern Matching Functions

Function	Parameters	Description
@zeroOrMoreWS	@zeroOrMoreWS(str, rem) *str* must be a string expression. *rem* is an optional argument. The type of *rem* is irrelevant but it must be a VAL object.	This function always returns true. It matches zero or more "whitespace" characters at the beginning of str. Whitespace includes spaces, newlines, tabs, and certain other special characters. Note that this function does not return the matched string. To return matched whitespace characters, use @zeroOrMoreCset.
@oneOrMoreWS	@oneOrMoreWS(str, rem) *str* must be a string expression. *rem* is an optional argument. The type of *rem* is irrelevant but it must be a VAL object. This function stores a result into this object.	This function returns true if it matches at least one whitespace character. If it returns true, it also returns any characters following the leading whitespace characters in the *rem* variable.
@wsOrEOS	@wsOrEOS(str, rem) *str* must be a string expression. *rem* is an optional argument. The type of *rem* is irrelevant but it must be a VAL object. This function stores a result into this object.	This function succeeds if there is whitespace at the beginning of *str* or if *str* is the empty string. If it succeeds and there is leading whitespace, this function returns the remaining characters in *rem*. If this function succeeds and the string was empty, this function returns the empty string in *rem*. This function fails if the string is not empty and it begins with non-whitespace characters.

Table 10: Miscellaneous Compile-time Pattern Matching Functions

Function	Parameters	Description
@wsThenEOS	@wsThenEOS(str) *str* must be a string expression.	This function returns true if *str* contains zero or more whitespace characters followed by the end of the string. If fails if there are any other characters in the string.
@peekWS	@peekWS(str, rem) *str* must be a string expression. *rem* is an optional argument. The type of *rem* is irrelevant but it must be a VAL object. This function stores a result into this object.	This function returns true if the next character in *str* is a whitespace character. This function returns a copy of *str* in *rem* if it is successful.
@eos	@eos(str) *str* must be a string expression.	This function returns true if and only if *str* is the empty string.

H.7 HLA Information and Symbol Table Functions

The symbol table functions provide access to information in HLA's internal symbol table database. These functions are particularly useful within macros to determine how to generate code for a particular macro parameter.

Table 11: HLA Information and Compile-time Symbol Table Information Functions

Function	Parameters	Description
@name	@name(identifier) *identifier* must be a defined symbol in the HLA program.	This function returns a string specifying the name of the specified identifier. The name this function returns is computed after macro and text constant expansion. This function is useful mainly in macros to determine the name of a macro parameter.

Table 11: HLA Information and Compile-time Symbol Table Information Functions

Function	Parameters	Description
@type	@type(identifier) *identifier* must be a defined symbol in the HLA program.	This function returns a *dword* constant that should be unique for any given type in the program. You can use this to compare the types of two different objects. For guaranteed uniqueness, see @typeName.
@typeName	@typeName(identifier) *identifier* must be a defined symbol in the HLA program.	This function returns a string that specifies the type of of the parameter.
@pType	@pType(identifierOrExpression) *identifierOrExpression* may be a defined symbol in the HLA program or a constant expression.	This function returns a small numeric constant that specifies the primitive type of the object. See the ptXXXXX constants in the HLA.HHF header file for the actual values this function returns.
@class	@class(identifierOrExpression) *identifierOrExpression* may be a defined symbol in the HLA program or a constant expression.	This function returns a small numeric constant that classifies the identifier or expression as to whether it is a constant, VAL, variable, parameter, static object, procedure, etc. See the cXXXX constants in the HLA.HHF header file for a list of the possible return values.
@size	@size(identifierOrExpression) *identifierOrExpression* may be a defined symbol in the HLA program or a constant expression.	This function returns the size, in bytes of the specified object.
@offset	@offset(identifier) *identifier* must be a defined VAR or parameter symbol in the HLA program.	This function returns a numeric constant providing the offset into a procedure's activation record for a VAR or parameter object.
@staticName	@staticName(identifier) *identifier* must be a defined static, procedure, method, iterator, or external symbol in the HLA program.	This function returns a string that specifies the internal name that HLA uses for the object.

Table 11: HLA Information and Compile-time Symbol Table Information Functions

Function	Parameters	Description
@lex	@lex(identifier) *identifier* must be a defined symbol in the HLA program.	This function returns a small integer constant the specifies the static nesting level for the specified symbol. All symbols appearing in the main program have a lex level of zero. Symbols you define in procedures within the main program have a lex level of one. Higher lex level values are possible if you define procedures inside procedures.
@isExternal	@isExternal(identifier) *identifier* must be a defined symbol in the HLA program.	This function returns true if the specified identifier is an external symbol. Note that this function will return true if the symbol is defined external and a declaration for the symbol appears later in the code.
@arity	@arity(identifier) *identifier* must be a defined symbol in the HLA program.	This function returns zero if the specified symbol is not an array. It returns a small integer constant denoting the number of dimensions if the identifier is an array object.
@dim	@dim(identifier) *identifier* must be a defined symbol in the HLA program.	If the specified identifier is an array object, this function returns an array constant with one element for each dimension in the array. Each element of this array constant specifies the number of elements for each dimension of the array. If the identifier is not an array object, this function returns an array with a single element and that element's value will be zero.
@elements	@elements(identifier) *identifier* must be a defined symbol in the HLA program.	This function returns the number of elements in an array object. If the specified identifier is not an array object, this function returns zero. For multi-dimensional arrays, this function returns the product of each of the dimensions.

Table 11: HLA Information and Compile-time Symbol Table Information Functions

Function	Parameters	Description
@elementSize	@elementSize(identifier) *identifier* must be a defined symbol in the HLA program.	This function returns the size, in bytes, of an element of the specified array.
@defined	@defined(identifier) *identifier* must be a defined symbol in the HLA program.	This function returns true if the specified symbol is defined prior to that point in the program.
@pClass	@pClass(identifier) *identifier* must be a parameter in the current procedure of an HLA program.	This function returns a small integer constant specifying the parameter passing mechanism for the specified parameter. The HLA.HHF header file defines the return values for this function (see the XXXX_pc constant declarations).
@isConst	@isConst(identifierOrExpression) *identifierOrExpression* may be a defined symbol in the HLA program or a constant expression.	This function returns true if the expression is a constant expression that HLA can evaluate at that point in the program.
@isReg	@isReg(Expression) *Expression* may be arbitrary text, but it is typically a register object.	This function returns true if the operand corresponds to an 80x86 general purpose register.
@isReg8	@isReg8(Expression) *Expression* may be arbitrary text, but it is typically a register object.	This function returns true if the operand corresponds to an eight-bit 80x86 general purpose register.
@isReg16	@isReg16(Expression) *Expression* may be arbitrary text, but it is typically a register object.	This function returns true if the operand corresponds to a 16-bit 80x86 general purpose register.

Table 11: HLA Information and Compile-time Symbol Table Information Functions

Function	Parameters	Description
@isReg32	@isReg32(Expression) *Expression* may be arbitrary text, but it is typically a register object.	This function returns true if the operand corresponds to a 32-bit 80x86 general purpose register.
@isFReg	@isFReg(Expression) *Expression* may be arbitrary text, but it is typically a register object.	This function returns true if the operand corresponds to an FPU register.
@isMem	@isMem(Expression) *Expression* may be arbitrary text, but it is typically a memory object (including various addressing modes).	This function returns true if the specified parameter corresponds to a legal 80x86 memory address.
@isClass	@isClass(Text) *Text* may be arbitrary text, but it is typically a class object.	This function returns true if the text parameter is a class name or a class object.
@isType	@isClass(Text) *Text* may be arbitrary text, but it is typically a type name.	This function returns true if the specified text is a type identifier.

Table 11: HLA Information and Compile-time Symbol Table Information Functions

Function	Parameters	Description
@section	@section	This function returns a 32-bit value that specifies which portion of the program HLA is currently processing. This information is mainly useful in macros to determine where a macro expansion is taking place. This function returns the following bit values: Bit 0: Currently in the CONST section. Bit 1: Currently in the VAL section. Bit 2: Currently in the TYPE section. bit 3: Currently in the VAR section. bit 4: Currently in the STATIC section. bit 5: In the READONLY section. Bit 6: In the STORAGE section. Bit 7: Currently in the DATA section. Bits 8-11: reserved. Bit 12: Processing statements in main. Bit 13: Statements in a procedure. Bit 14: Statements in a method. Bit 15: Statements in an iterator. Bit 16: Statements in a macro. Bit 17: Statements in a keyword macro. Bit 18: In a Terminator macro. Bit 19: Statements in a Thunk. Bits 20-22: reserved. Bit 23: Processing statements in a Unit. Bit 24: Statements in a Program. Bit 25: Processing a Record declaration. Bit 26: Processing a Union declaration. Bit 27: Processing a Class declaration. Bit 28: Processing a Namespace declaration.

Table 11: HLA Information and Compile-time Symbol Table Information Functions

Function	Parameters	Description
@curLex	@curLex	Returns the current lex level of this statement within the program.
@curOffset	@curOffset	Returns the current offset into the activation record. This is the offset of the last VAR object declared in the current program/procedure.
@curDir	@curDir	Returns +1 if processing parameters, -1 otherwise. This corresponds to whether offsets are increasing or decreasing in an activation record during compilation. This function also returns +1 when processing fields in a record or class; it returns zero when processing fields of a union.
@addofs1st	@addofs1st	This function returns true when processing local variables, it returns false when processing parameters and record/class/union declarations.
@lastObject	@lastObject	This function returns a string containing the name of the last macro object processed.
@lineNumber	@lineNumber	This function returns an *uns32* value specifying the current line number in the file.

H.8 Compile-Time Variables

The HLA compile-time variables are special symbols that not only return values, but allow you to modify their internal values as well. You may use these symbols exactly like any VAL object in your program with the exception that you cannot change the type (generally *int32* or *boolean*) of these objects. By changing the values of these *pseudo-variables*, you can affect the way HLA generates code in your programs.

Table 12: HLA Compile-time Variables

Pseudo-Variable	Description
@parmOffset	This variable specifies the starting offset for parameters in a function. This should be eight for most procedures. If you change this value, HLA's automatic code generation for procedure calls may fail.
@localOffset	This variable specifies the starting offset for local variables in a procedure. This is typically zero. If you change this value, it will affect the offsets of all local symbols in the activation record, hence you should modify this value only if you really know what you're doing (and have a good reason for doing it).
@enumSize	This variable specifies the size (in bytes) of enum objects in your program. By default, this value is one. If you want word or dword sized enum objects you can change this variable to two or four. Other values may create problems for the compiler.
@minParmSize	This variable specifies the minimum number of bytes for each parameter. Under Windows, this should always be four.
@bound	This boolean variable, whose default value is true, controls the compilation of the BOUND instruction. If this variable contains false, HLA does not compile BOUND instructions. You can set this value to true or false throughout your program to control the emission of bound instructions.
@into	This boolean variable, whose default value is false, controls the compilation of the INTO instruction. If this variable contains false, HLA does not compile INTO instructions. You can set this value to true or false throughout your program to control the emission of INTO instructions.
@trace	Enables HLA's statement tracing facilities. See the separate appendix for details.
@exceptions	If true (the default) then HLA uses the full exception handling package in the HLA Standard Library. If false, HLA uses a truncated version. This allows programmers to write their own exception handling code.

H.9 Miscellaneous Compile-Time Functions

This last category contains various functions and objects that are quite useful in compile-time programs.

Table 13: Miscellaneous HLA Compile-time Functions and Objects

Function	Parameters	Description
@text	@text(str) *str* must be a string constant expression.	This function expands the string constant in-line as text, replacing this function invocation with the specified text.
@eval	@eval(text) *text* is an arbitrary sequence of textual characters.	This function is useable only within macro invocation parameter lists. It immediately evaluates the text object and passes the resulting text as the parameter to the macro. This provides eager evaluation capabilities for macro parameters.
@string:*identifier*	@string:*identifier* *identifier* must be a *text* constant.	This function returns a string constant corresponding to the string data in the specified identifer. It does not otherwise affect the value or type of *identifier.*
@toString:*identifier*	@string:*identifier* *identifier* must be a *text* constant.	This function converts the type of *identifier* from *text* to *string* and then returns the value of that string object.
@global:*identifier*	@string:*identifier* *identifier* must be an HLA identifier declared outside the current namespace.	@global is legal only within a namespace declaration. It provides access to identifiers outside the namespace.

